PUCRS | ESCOLA POLITÉCNICA

Projeto e Otimização de Algoritmos

Divisão e Conquista

MARISTA

Prof. Rafael Scopel

Algoritmos de Divisão e Conquista

- Divisão e conquista se refere a uma classe de algoritmos na qual a entrada é quebrada em várias partes.
- Cada parte é resolvida de forma recursiva e então combinada com as demais para dar a solução final.
- A análise do tempo de recursão de um algoritmo de divisão e conquista normalmente envolve resolver uma relação de recorrência.
- A solução da relação de recorrência cria um limite no tempo da execução recursiva em termos do tempo de execução das menores partes da recursão.
- Algoritmos de divisão e conquista são utilizados em muitos domínios como:
  - ordenar valores;
  - computação de distâncias mínimas;
  - encontrar os dois pontos mais próximos num plano;
  - multiplicar dois números inteiros
  - suavizar um sinal com ruído, ...

Algoritmos de Divisão e Conquista

O algoritmo de divisão e conquista possuí 3 etapas:

1. **Dividir**: Se a entrada é menor do que um limite (por exemplo, um ou dois elementos), resolve o problema utilizando um método direto e retornando a solução obtida. Senão, divida os dados de entrada em dois os mais conjuntos disjuntos.
2. **Conquistar**: Recursivamente resolva os problemas associados com cada subconjunto da etapa anterior.
3. **Combinar**: Combine as subsoluções da etapa anterior para gerar a solução final do problema.

Algoritmos de Divisão e Conquista

- Dada uma função para calcular n entradas, a estratégia de dividir e conquistar sugere dividir as entradas em k subconjuntos distintos, $1 < k \leq n$, produzindo k subproblemas.
- Esses subproblemas devem ser resolvidos e então um método deve ser encontrado para combinar subsoluções em uma solução do todo.
- Se os subproblemas ainda forem relativamente grandes, então a estratégia de dividir para conquistar poderá ser reaplicada.
- Freqüentemente, os subproblemas resultantes de um problema de dividir para conquistar são do mesmo tipo que o problema original.
- Para esses casos, a reaplicação do princípio de dividir e conquistar é naturalmente expressa por um algoritmo recursivo.
- Agora, subproblemas cada vez menores do mesmo tipo são gerados até que, eventualmente, subproblemas que são pequenos o suficiente para serem resolvidos sem divisão.

Algoritmos de Divisão e Conquista

- Suponha que consideramos a estratégia de dividir para conquistar quando ela divide a entrada em dois subproblemas do mesmo tipo que o problema original.
- Esta divisão é típica de muitos dos problemas em computação.
- Podemos escrever uma abstração de controle que espelhe a aparência de um algoritmo baseado em dividir e conquistar.
- Por abstração de controle entendemos um procedimento cujo fluxo de controle é claro, mas cujas operações primárias são especificadas por outros procedimentos cujos significados precisos são deixados indefinidos.
- O algoritmo $DAndC$ é inicialmente invocado como $DAndC(P)$, onde P é o problema a ser resolvido.
- $Small(P)$ é uma função com valor booleano que determina se a entrada o tamanho é suficiente para que a resposta possa ser calculada sem divisão.
- Se a condição for verdadeira, a função S é invocada.

```
Algorithm DAndC(P)
{
    if Small(P) then return S(P);
    else
    {
        divide P into smaller instances P_1, P_2, ..., P_k, k ≥ 1;
        Apply DAndC to each of these subproblems;
        return Combine(DAndC(P_1),DAndC(P_2),...,DAndC(P_k));
    }
}
```

- Caso contrário, o problema P é dividido em subproblemas menores.
- Esses subproblemas $P_1, P_2, \dots, P_k$ são resolvidos por aplicações recursivas de $DAndC$.
- $Combine$ é uma função que determina a solução de P usando as soluções dos k subproblemas.
- Se o tamanho de P for n e os tamanhos dos k subproblemas forem $n_1, n_2, \dots, n_k$, respectivamente, então o o tempo de computação de DAndC é descrito pela relação de recorrência:

Algoritmos de Divisão e Conquista

```
Algorithm DAndC(P)
{
    if Small(P) then return S(P);
    else
    {
        divide P into smaller instances P_1, P_2, ..., P_k, k ≥ 1;
        Apply DAndC to each of these subproblems;
        return Combine(DAndC(P_1),DAndC(P_2),...,DAndC(P_k));
    }
}
```

- Se o tamanho de P for n e os tamanhos dos k subproblemas forem $n_1, n_2, \ldots, n_k$, respectivamente, então o o tempo de computação de DAndC é descrito pela relação de recorrência:

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n_1) + T(n_2) + \cdots + T(n_k) \quad + \quad f(n) & \text{otherwise} \end{cases}$$

- onde $T(n)$ é o tempo para $DAndC$ em qualquer entrada de tamanho n.
- $g(n)$ é o tempo para calcular a resposta diretamente para entradas pequenas.
- A função $f(n)$ é o tempo para dividir P e combinar as soluções dos subproblemas.

- Para algoritmos baseados em divisão e conquista que produzem subproblemas do mesmo tipo, assim como o problema original, é muito natural primeiro descrever tais algoritmos usando recursão.

Algoritmos de Divisão e Conquista: Merge-Sort

- O algoritmo de ordenação Merge-Sort é um clássico exemplo de divisão e conquista.
- Para ordenar uma sequência S com n elementos, o algoritmo de Merge-Sort procede da seguinte maneira:
  1. Dividir:
     - Se S tem zero ou 1 elemento, retorne S imediatamente.
     - Se $|S| \geq 2$, remova todos os elementos de S e coloque-os em duas sequencias, S_1 e S_2, cada uma contendo aproximadamente metade dos elementos de S. Ou seja, S_1 contem $\lfloor n/2 \rfloor$ e S_2 contem $\lceil n/2 \rceil$ elementos de S.
  2. Conquistar:
     - Recursivamente ordene as sequências de S_1 e S_2.
  3. Combinar:
     - Coloque todos os elementos novamente em S combinando as sequências ordenadas S_1 e S_2 numa sequência final ordenada.

MERGE-SORT(S)

IF (list S has one element)

  RETURN S.

Divide the list into two halves S_1 and S_2

$S_1 \leftarrow$ MERGE-SORT(S_1). ⟵ $T(n/2)$

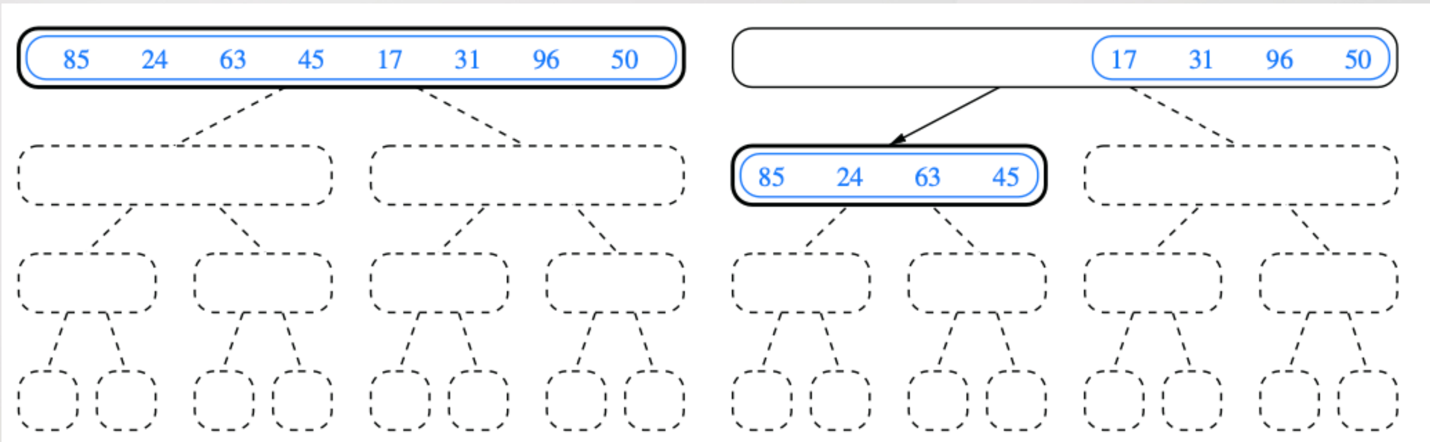
$S_2 \leftarrow$ MERGE-SORT(S_2). ⟵ $T(n/2)$

$S \leftarrow$ MERGE(S_1, S_2). ⟵ $\Theta(n)$

RETURN S.

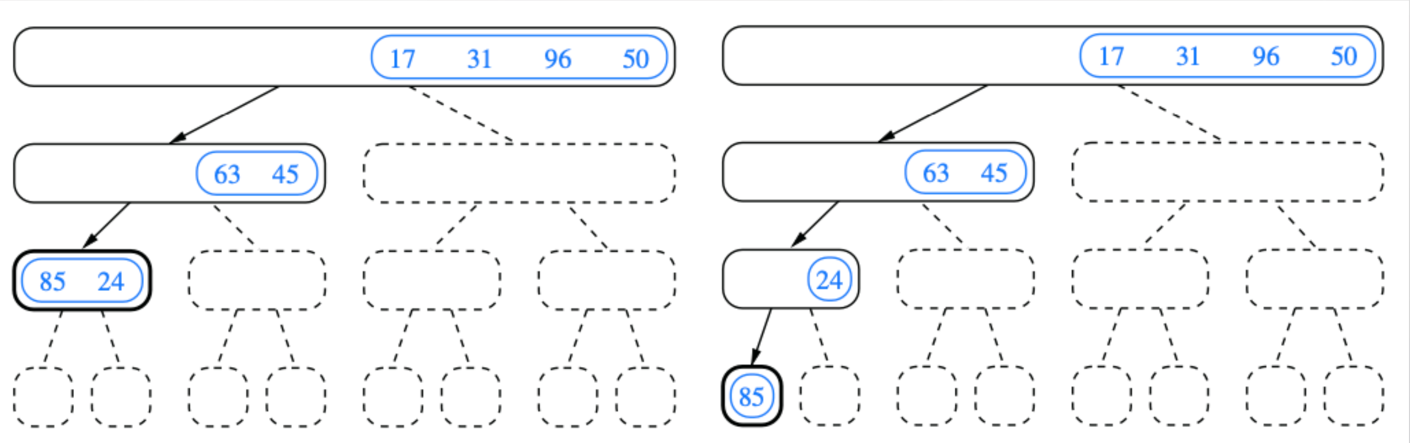
Algoritmos de Divisão e Conquista: Merge-Sort

- Podemos visualizar a execução do algoritmo Merge-Sort utilizando uma árvore binária T, chamada de **merge-sort tree**.
- Cada nodo em T representa uma chamada recursiva do algoritmo de merge-sort.
- Associamos com cada nodo v de T a sequência S que é processada pela chamada associada com v.
- Os filhos do nodo v são associados com chamadas recursivas que processam as subsequências S_1 e S_2 de S.
- Os nodos terminais (folhas) de T são associados com elementos individuais de S, correspondendo a instâncias do algoritmo que não fazem chamadas recursivas.



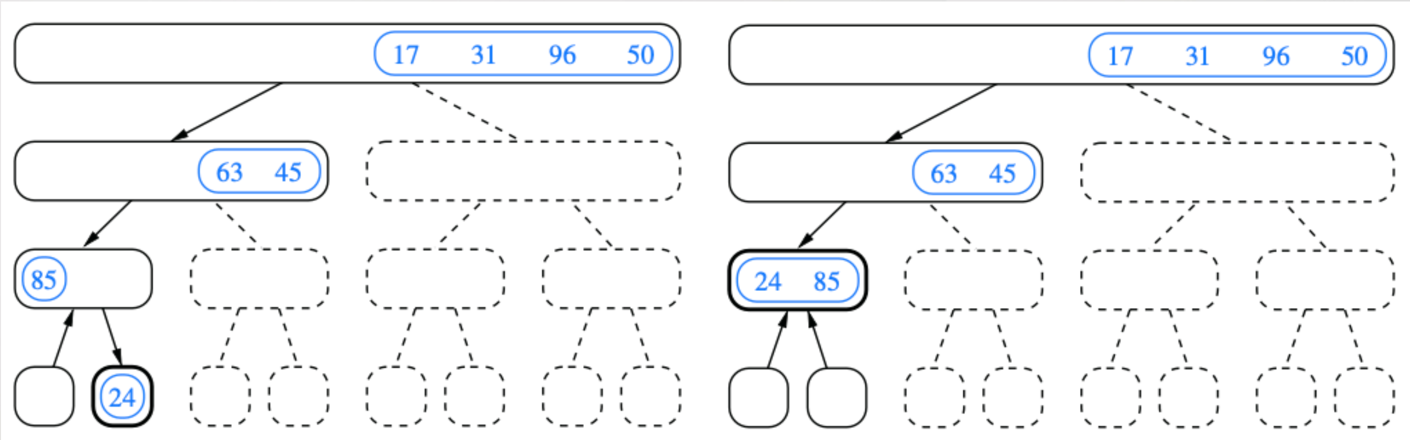
Algoritmos de Divisão e Conquista: Merge-Sort

- Podemos visualizar a execução do algoritmo Merge-Sort utilizando uma árvore binária T, chamada de **merge-sort tree**.
- Cada nodo em T representa uma chamada recursiva do algoritmo de merge-sort.
- Associamos com cada nodo v de T a sequência S que é processada pela chamada associada com v.
- Os filhos do nodo v são associados com chamadas recursivas que processam as subsequências S_1 e S_2 de S.
- Os nodos terminais (folhas) de T são associados com elementos individuais de S, correspondendo a instâncias do algoritmo que não fazem chamadas recursivas.



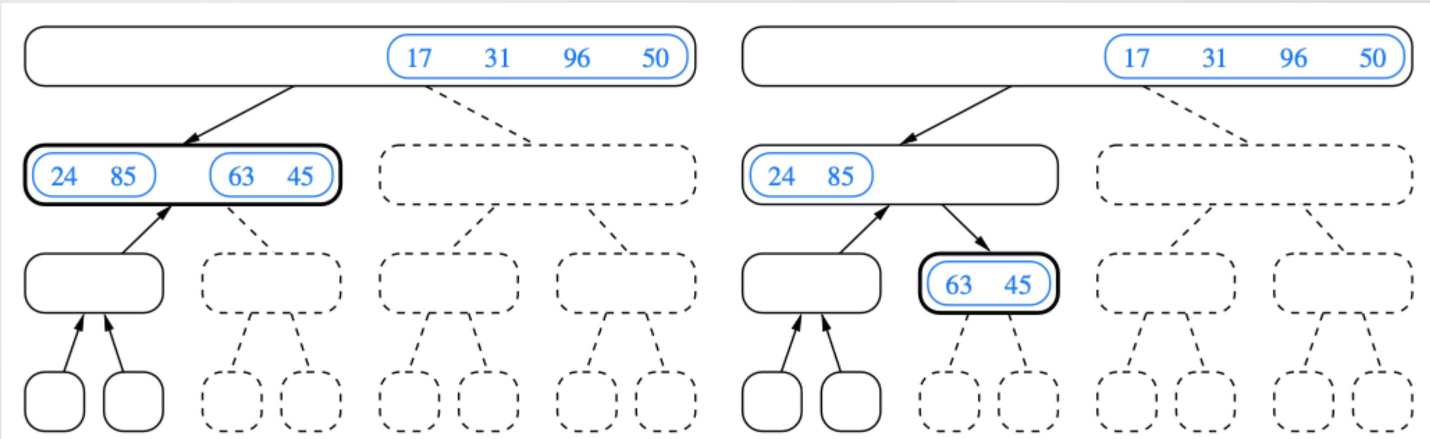
Algoritmos de Divisão e Conquista: Merge-Sort

- Podemos visualizar a execução do algoritmo Merge-Sort utilizando uma árvore binária T, chamada de **merge-sort tree**.
- Cada nodo em T representa uma chamada recursiva do algoritmo de merge-sort.
- Associamos com cada nodo v de T a sequência S que é processada pela chamada associada com v.
- Os filhos do nodo v são associados com chamadas recursivas que processam as subsequências S_1 e S_2 de S.
- Os nodos terminais (folhas) de T são associados com elementos individuais de S, correspondendo a instâncias do algoritmo que não fazem chamadas recursivas.



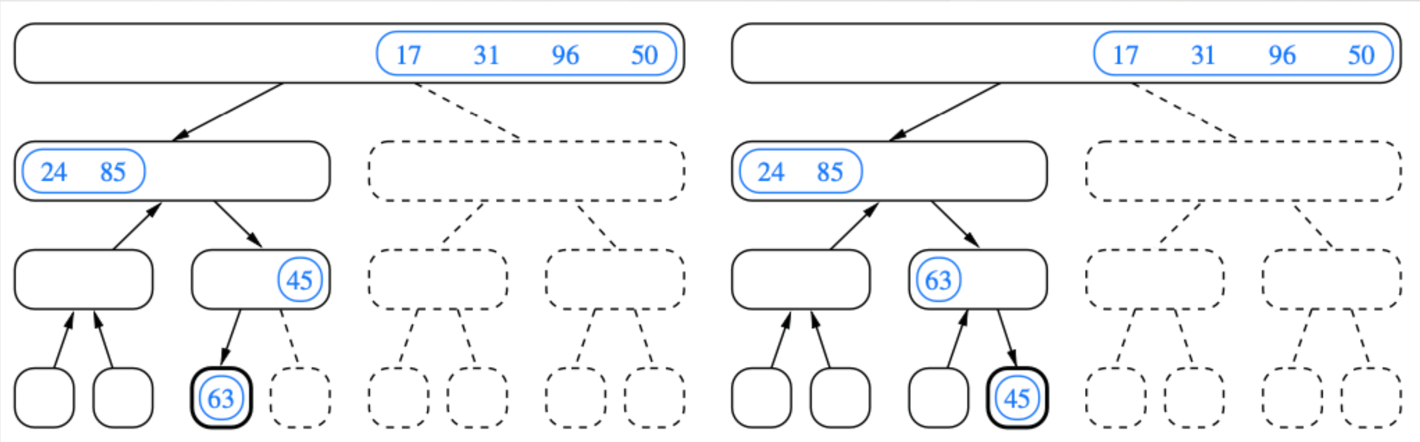
Algoritmos de Divisão e Conquista: Merge-Sort

- Podemos visualizar a execução do algoritmo Merge-Sort utilizando uma árvore binária T, chamada de **merge-sort tree**.
- Cada nodo em T representa uma chamada recursiva do algoritmo de merge-sort.
- Associamos com cada nodo v de T a sequência S que é processada pela chamada associada com v.
- Os filhos do nodo v são associados com chamadas recursivas que processam as subsequências S_1 e S_2 de S.
- Os nodos terminais (folhas) de T são associados com elementos individuais de S, correspondendo a instâncias do algoritmo que não fazem chamadas recursivas.



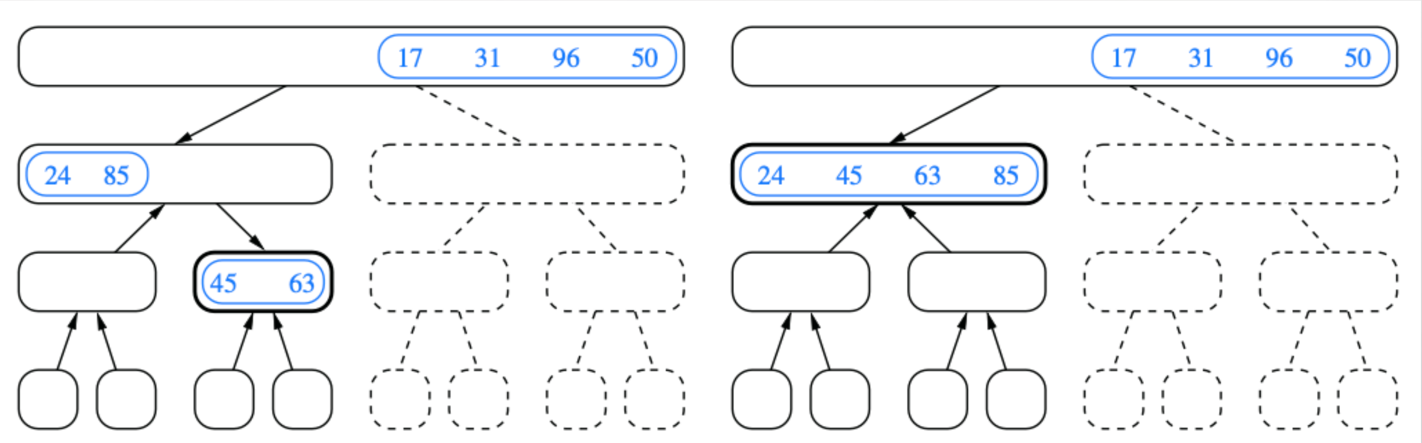
Algoritmos de Divisão e Conquista: Merge-Sort

- Podemos visualizar a execução do algoritmo Merge-Sort utilizando uma árvore binária T, chamada de **merge-sort tree**.
- Cada nodo em T representa uma chamada recursiva do algoritmo de merge-sort.
- Associamos com cada nodo v de T a sequência S que é processada pela chamada associada com v.
- Os filhos do nodo v são associados com chamadas recursivas que processam as subsequências S_1 e S_2 de S.
- Os nodos terminais (folhas) de T são associados com elementos individuais de S, correspondendo a instâncias do algoritmo que não fazem chamadas recursivas.



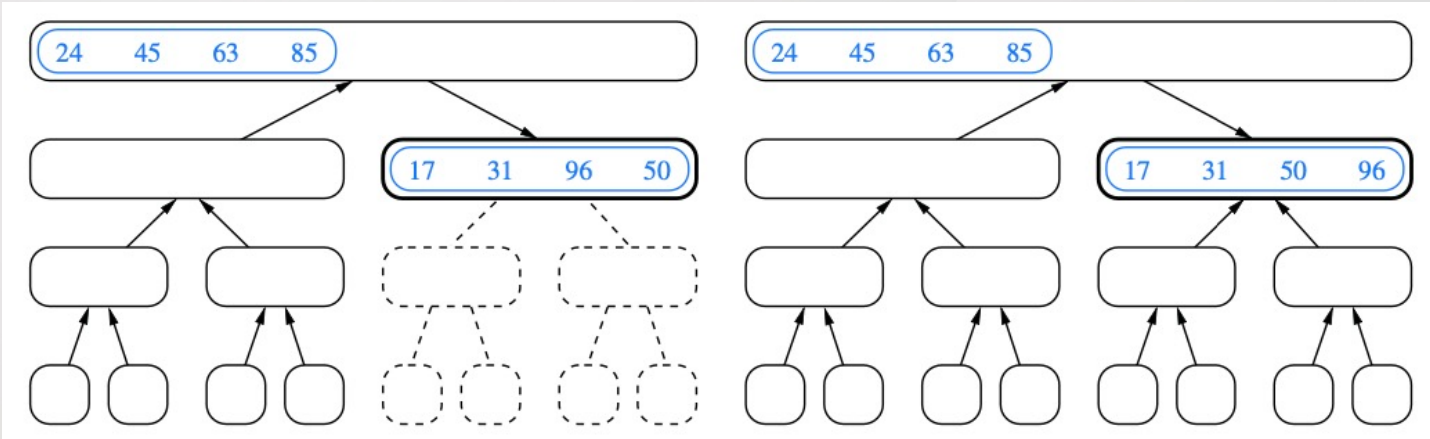
Algoritmos de Divisão e Conquista: Merge-Sort

- Podemos visualizar a execução do algoritmo Merge-Sort utilizando uma árvore binária T, chamada de **merge-sort tree**.
- Cada nodo em T representa uma chamada recursiva do algoritmo de merge-sort.
- Associamos com cada nodo v de T a sequência S que é processada pela chamada associada com v.
- Os filhos do nodo v são associados com chamadas recursivas que processam as subsequências S_1 e S_2 de S.
- Os nodos terminais (folhas) de T são associados com elementos individuais de S, correspondendo a instâncias do algoritmo que não fazem chamadas recursivas.



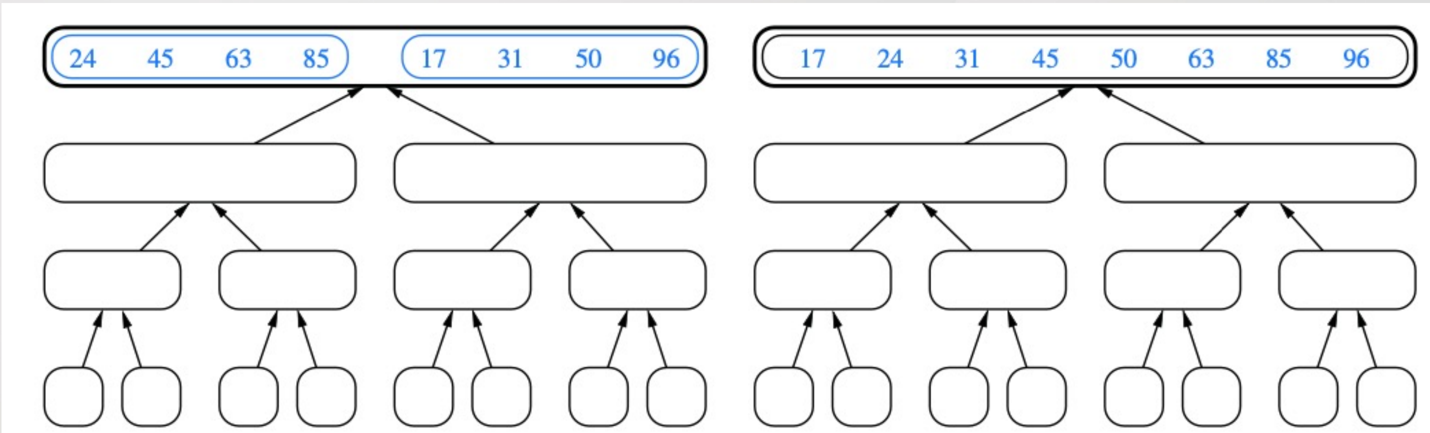
Algoritmos de Divisão e Conquista: Merge-Sort

- Podemos visualizar a execução do algoritmo Merge-Sort utilizando uma árvore binária T, chamada de **merge-sort tree**.
- Cada nodo em T representa uma chamada recursiva do algoritmo de merge-sort.
- Associamos com cada nodo v de T a sequência S que é processada pela chamada associada com v.
- Os filhos do nodo v são associados com chamadas recursivas que processam as subsequências S_1 e S_2 de S.
- Os nodos terminais (folhas) de T são associados com elementos individuais de S, correspondendo a instâncias do algoritmo que não fazem chamadas recursivas.



Algoritmos de Divisão e Conquista: Merge-Sort

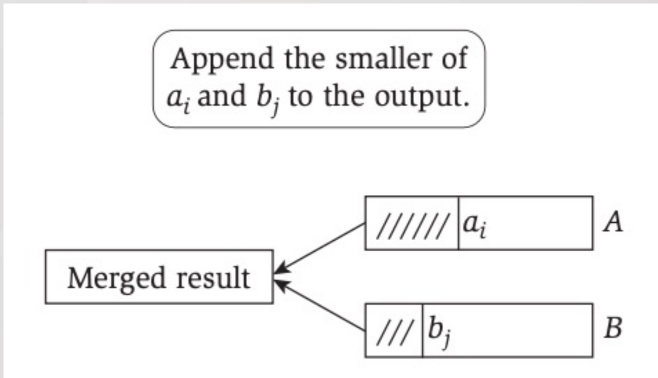
- Podemos visualizar a execução do algoritmo Merge-Sort utilizando uma árvore binária T, chamada de **merge-sort tree**.
- Cada nodo em T representa uma chamada recursiva do algoritmo de merge-sort.
- Associamos com cada nodo v de T a sequência S que é processada pela chamada associada com v.
- Os filhos do nodo v são associados com chamadas recursivas que processam as subsequências S_1 e S_2 de S.
- Os nodos terminais (folhas) de T são associados com elementos individuais de S, correspondendo a instâncias do algoritmo que não fazem chamadas recursivas.



Algoritmos de Divisão e Conquista: Merge-Sort

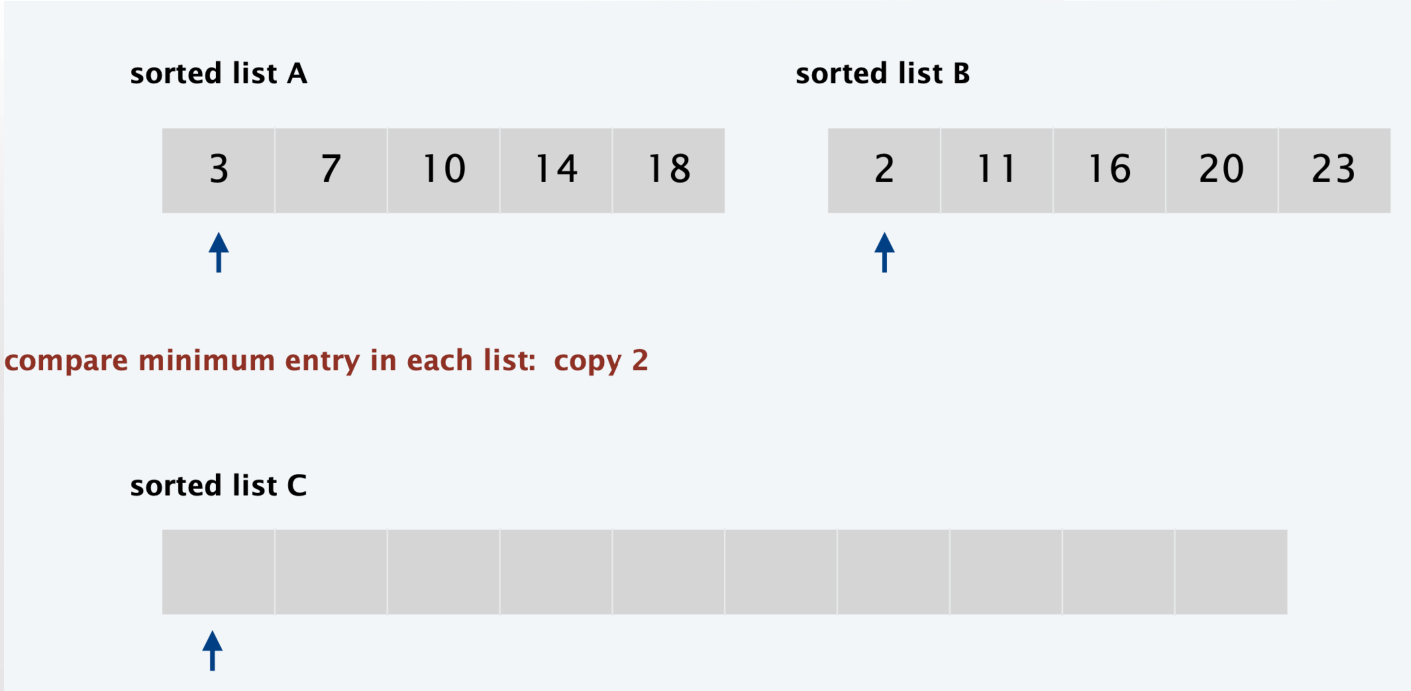
- A combinação (merge) das listas ordenadas S_1 e S_2 poderia ser feita diretamente, sem considerar o fato que estão ordenadas. Resultando num tempo de $O(n \log n)$ ou pior...
- Um desenho melhor de algoritmo é realizar a combinação como se fizéssemos manualmente.
- Suponha duas pilhas de cartas numeradas, cada uma ordenada de forma ascendente, e que desejamos organizá-las de numa única pilha de cartas ordenadas.
- Se você observar o topo de cada pilha de cartas, a carta com o menor valor deve ser adiciona na nova pilha de cartas.
- Agora uma nova carta está no topo da pilha, comparamos novamente as cartas das duas pilhas e colocamos a menor na nova pilha...
- Procedemos iterativamente até as duas pilhas iniciais estarem vazias...
- Com isso atingimos uma complexidade de $O(n)$, pois cada elemento escolhido é analisado uma única vez e removido da lista. A cada nova iteração temos um número menor de valores para procurar/comparar.

```
To merge sorted lists A = a_1, ..., a_n and B = b_1, ..., b_n:
  Maintain a Current pointer into each list, initialized to
    point to the front elements
  While both lists are nonempty:
    Let a_i and b_j be the elements pointed to by the Current pointer
    Append the smaller of these two to the output list
    Advance the Current pointer in the list from which the
      smaller element was selected
  EndWhile
  Once one list is empty, append the remainder of the other list
      to the output
```



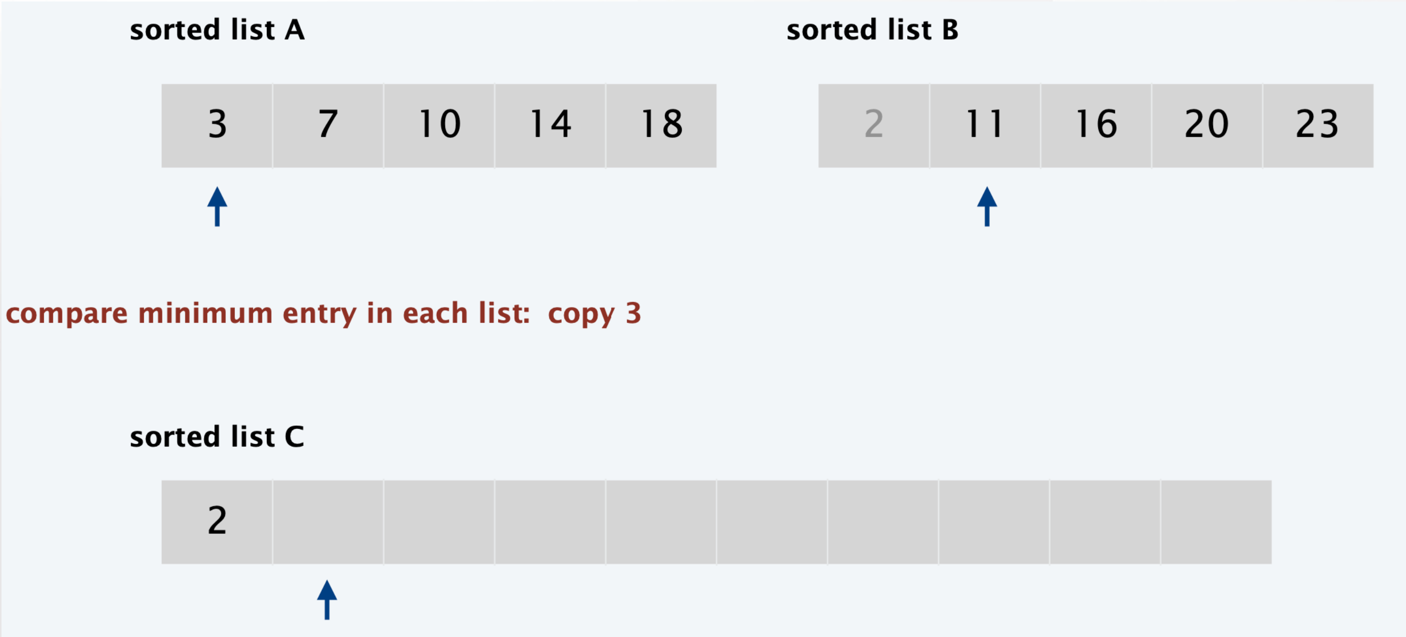
Algoritmos de Divisão e Conquista: Merge-Sort

- Dado duas listas ordenadas A e B, a combinação de ambas na lista C ocorre da seguinte maneira:



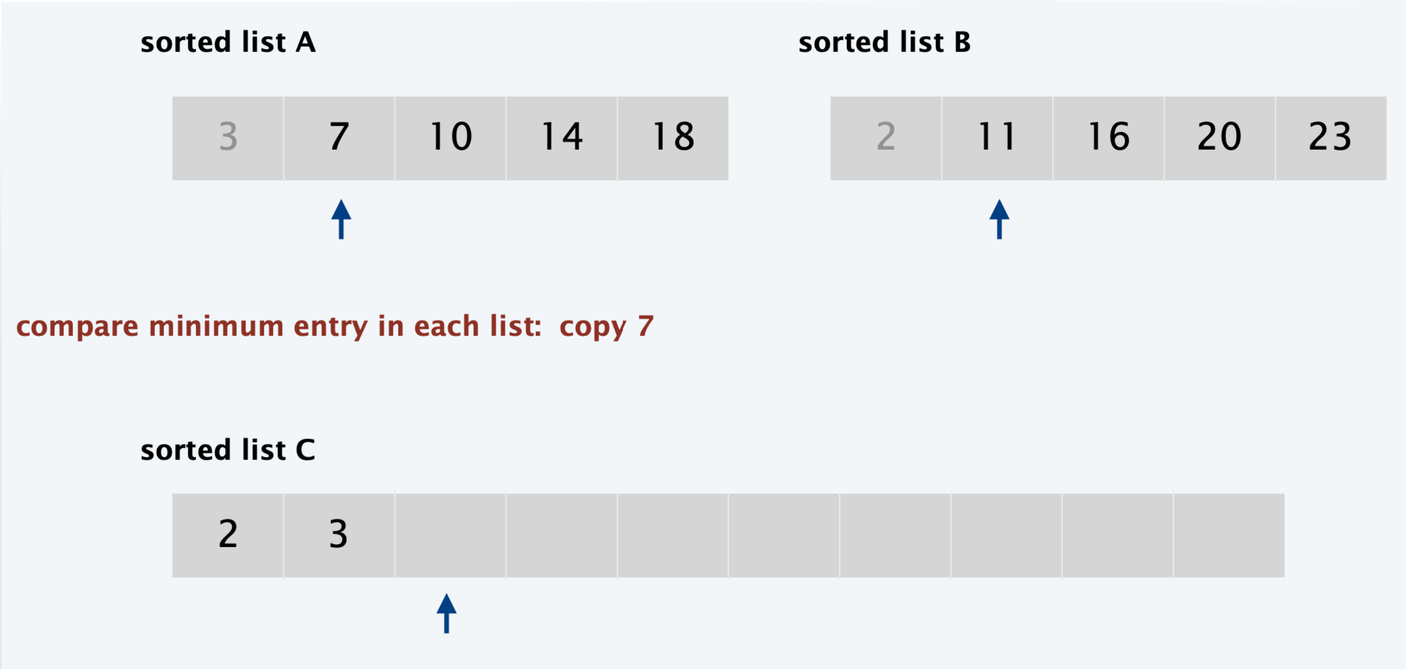
Algoritmos de Divisão e Conquista: Merge-Sort

- Dado duas listas ordenadas A e B, a combinação de ambas na lista C ocorre da seguinte maneira:



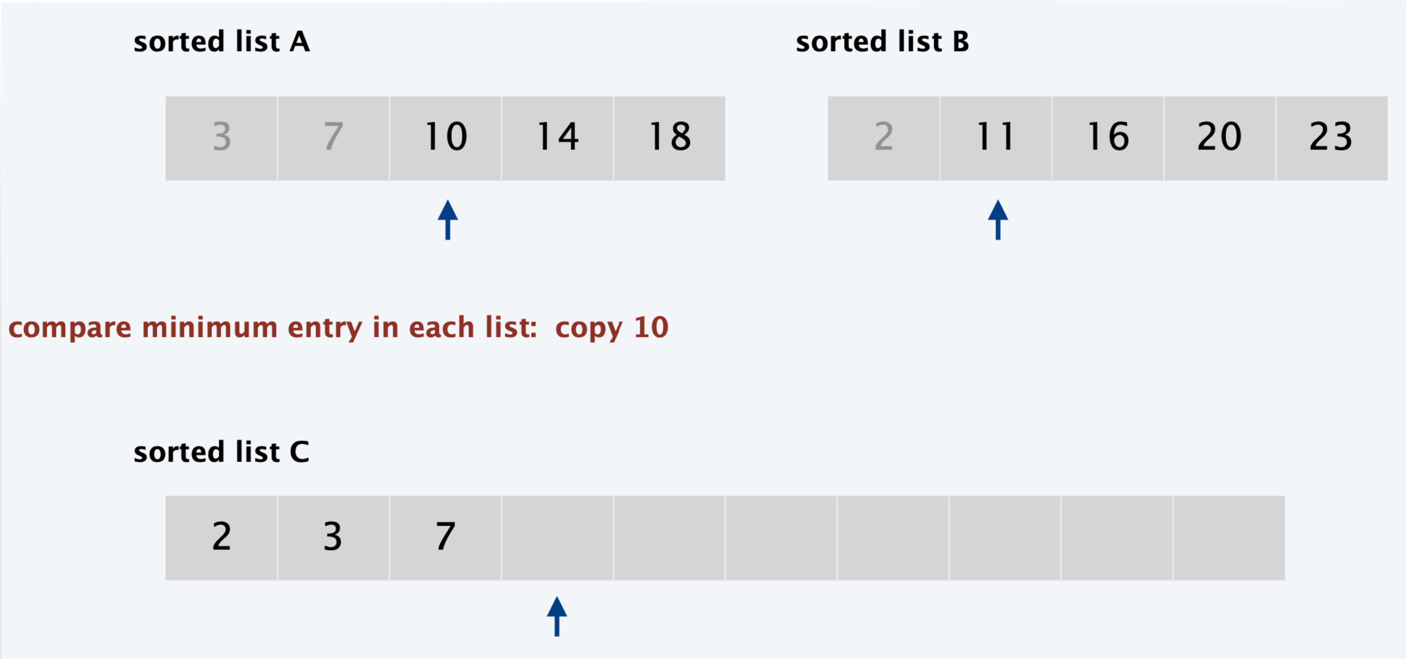
Algoritmos de Divisão e Conquista: Merge-Sort

- Dado duas listas ordenadas A e B, a combinação de ambas na lista C ocorre da seguinte maneira:



Algoritmos de Divisão e Conquista: Merge-Sort

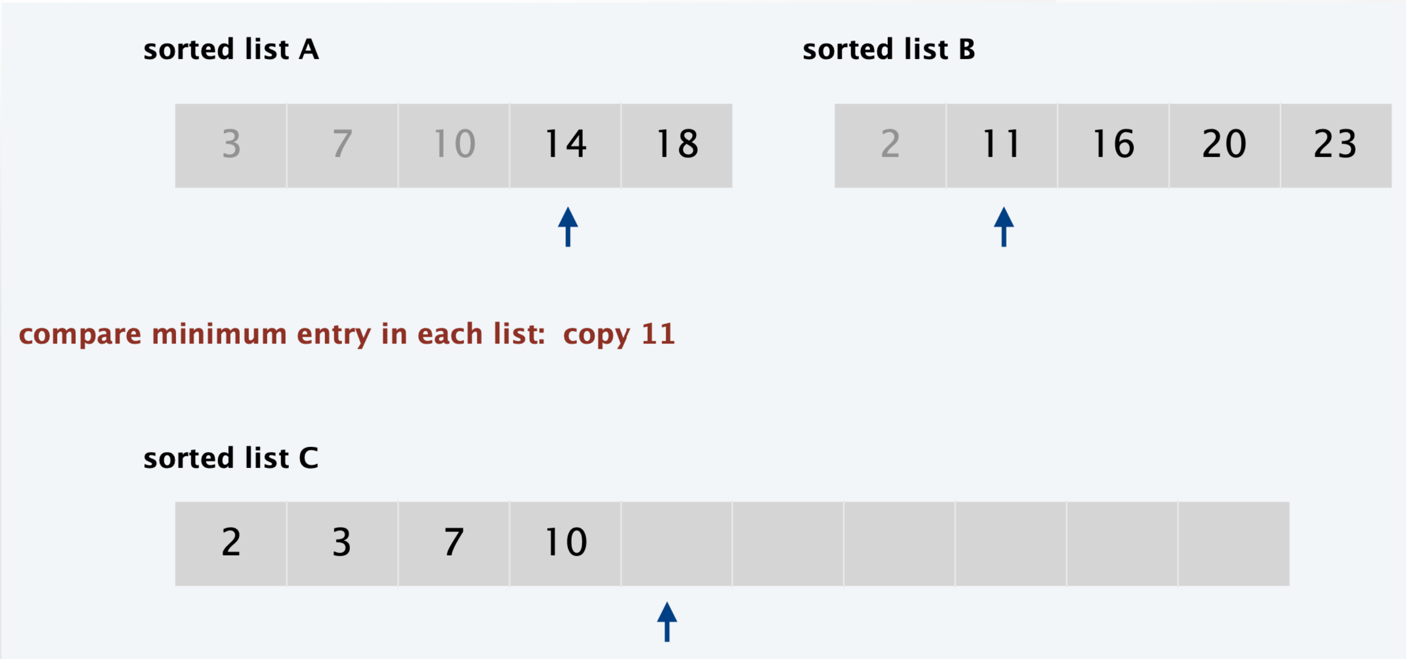
- Dado duas listas ordenadas A e B, a combinação de ambas na lista C ocorre da seguinte maneira:



PUCRS ESCOLA POLITÉCNICA

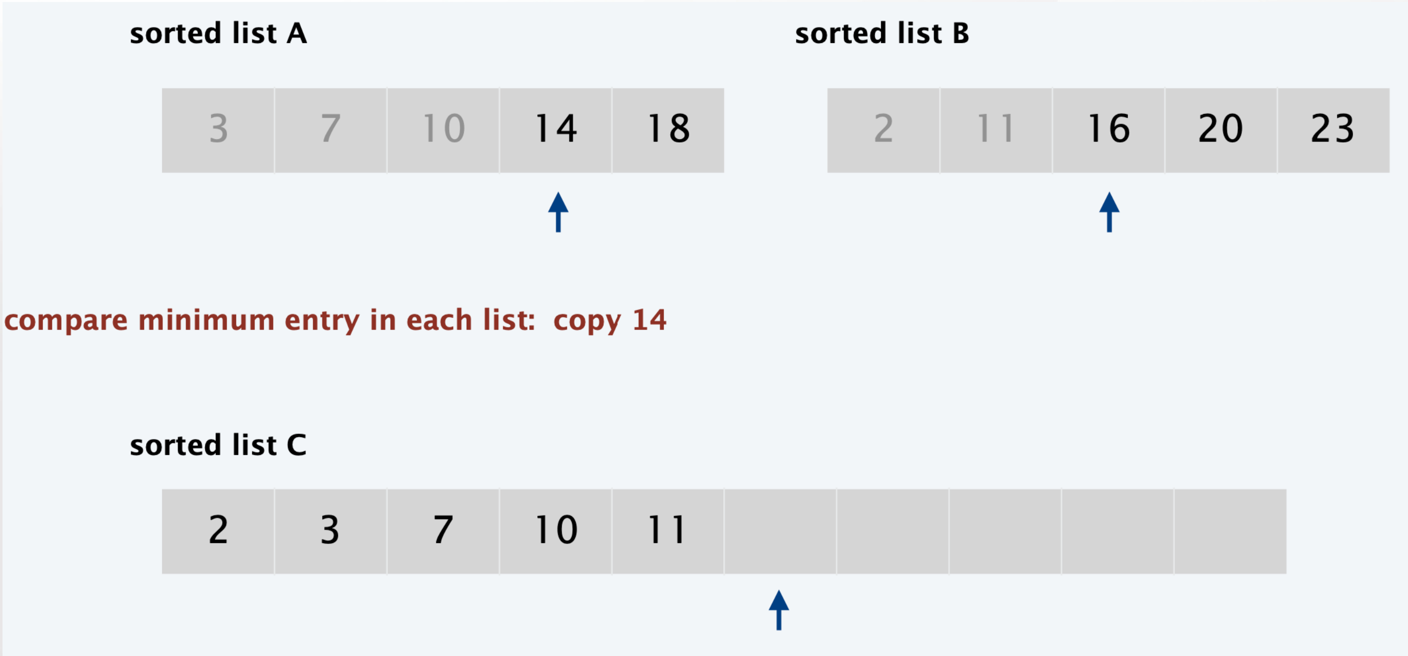
Algoritmos de Divisão e Conquista: Merge-Sort

- Dado duas listas ordenadas A e B, a combinação de ambas na lista C ocorre da seguinte maneira:



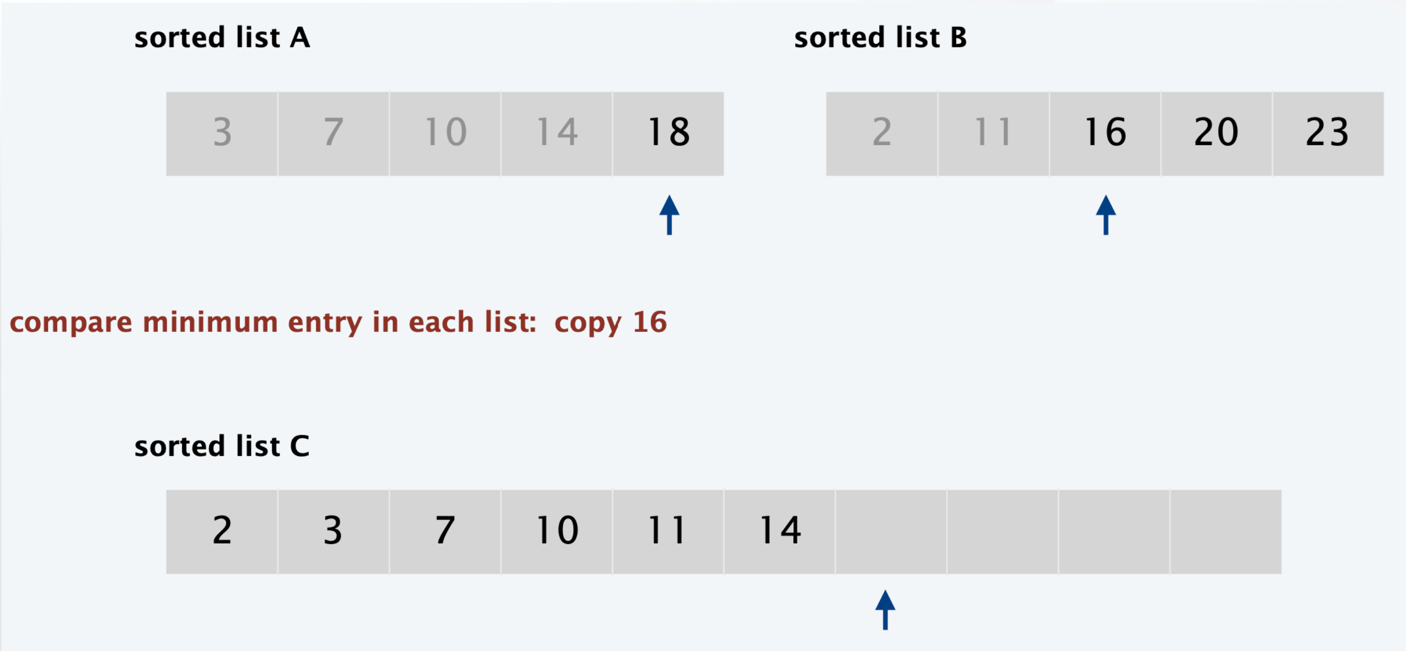
Algoritmos de Divisão e Conquista: Merge-Sort

- Dado duas listas ordenadas A e B, a combinação de ambas na lista C ocorre da seguinte maneira:



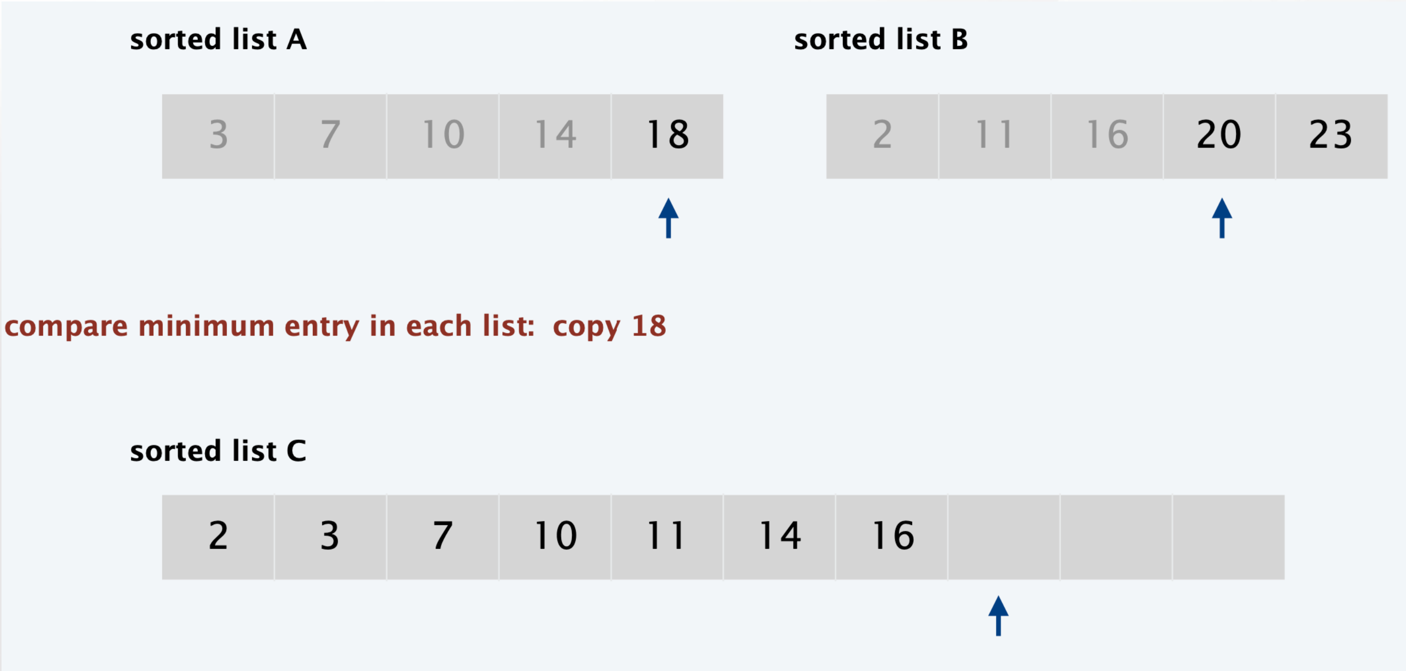
Algoritmos de Divisão e Conquista: Merge-Sort

- Dado duas listas ordenadas A e B, a combinação de ambas na lista C ocorre da seguinte maneira:



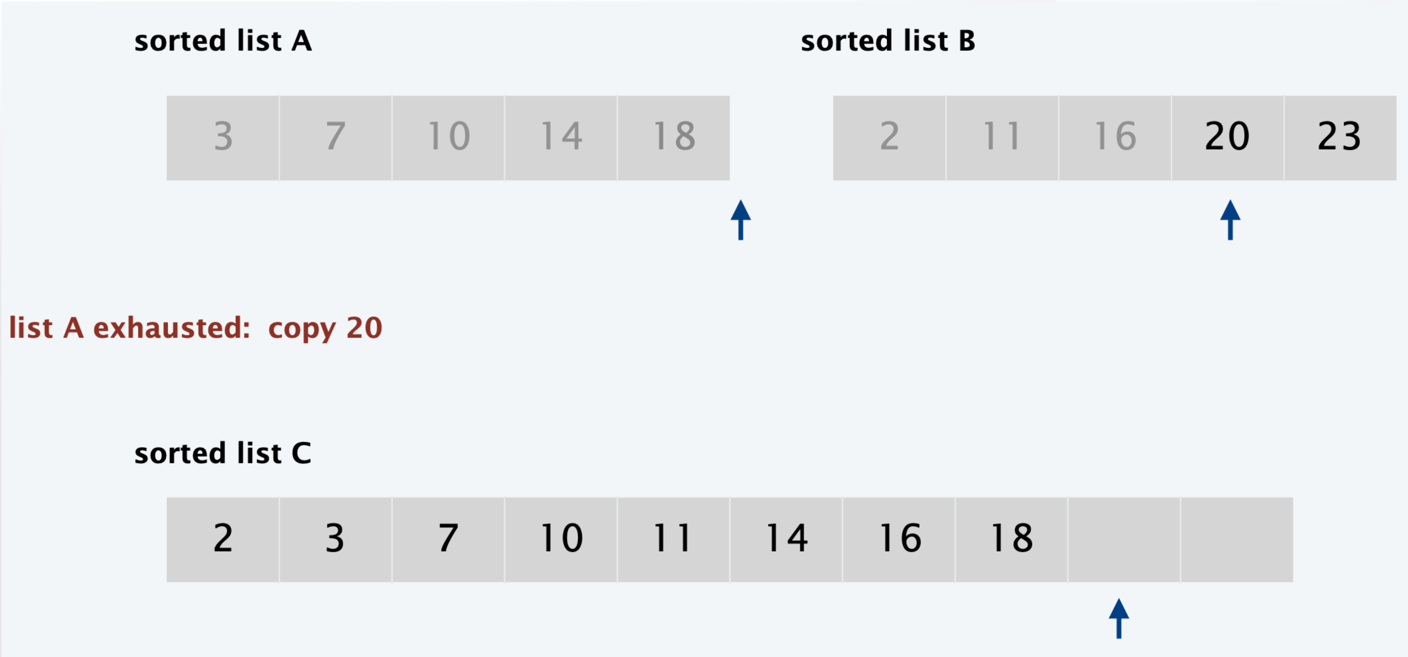
Algoritmos de Divisão e Conquista: Merge-Sort

- Dado duas listas ordenadas A e B, a combinação de ambas na lista C ocorre da seguinte maneira:



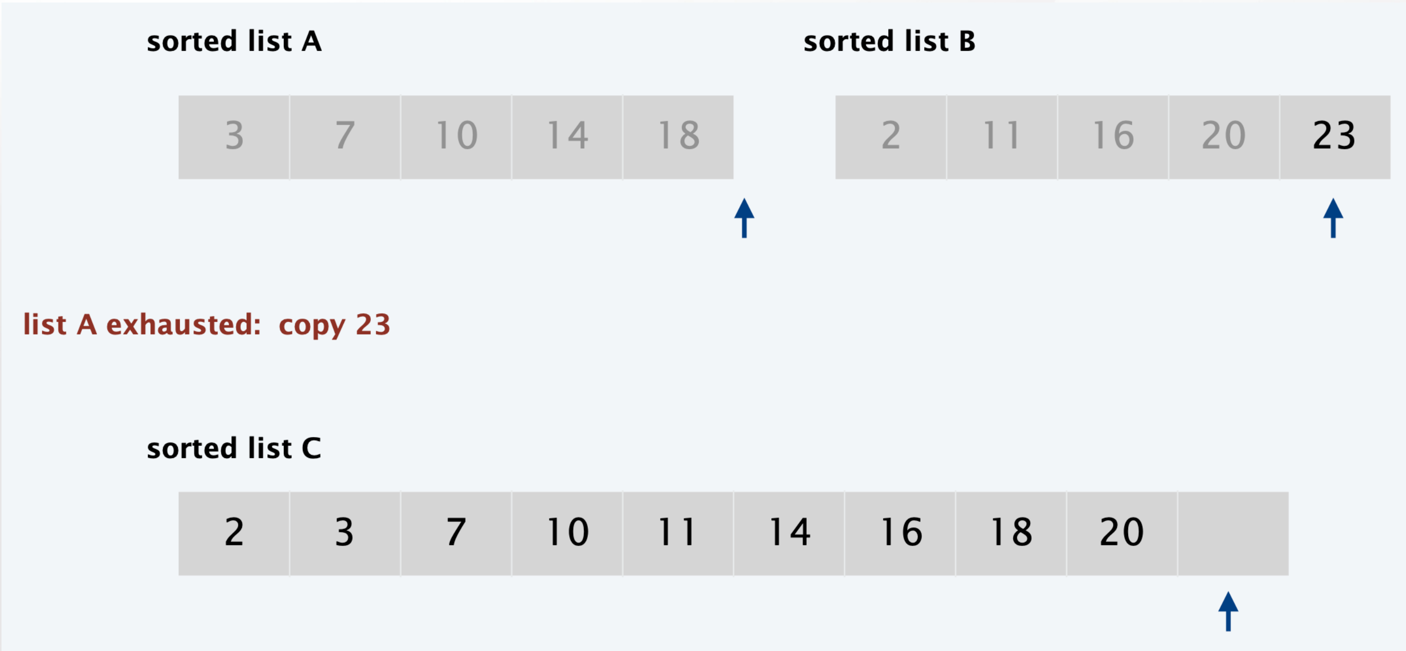
Algoritmos de Divisão e Conquista: Merge-Sort

- Dado duas listas ordenadas A e B, a combinação de ambas na lista C ocorre da seguinte maneira:



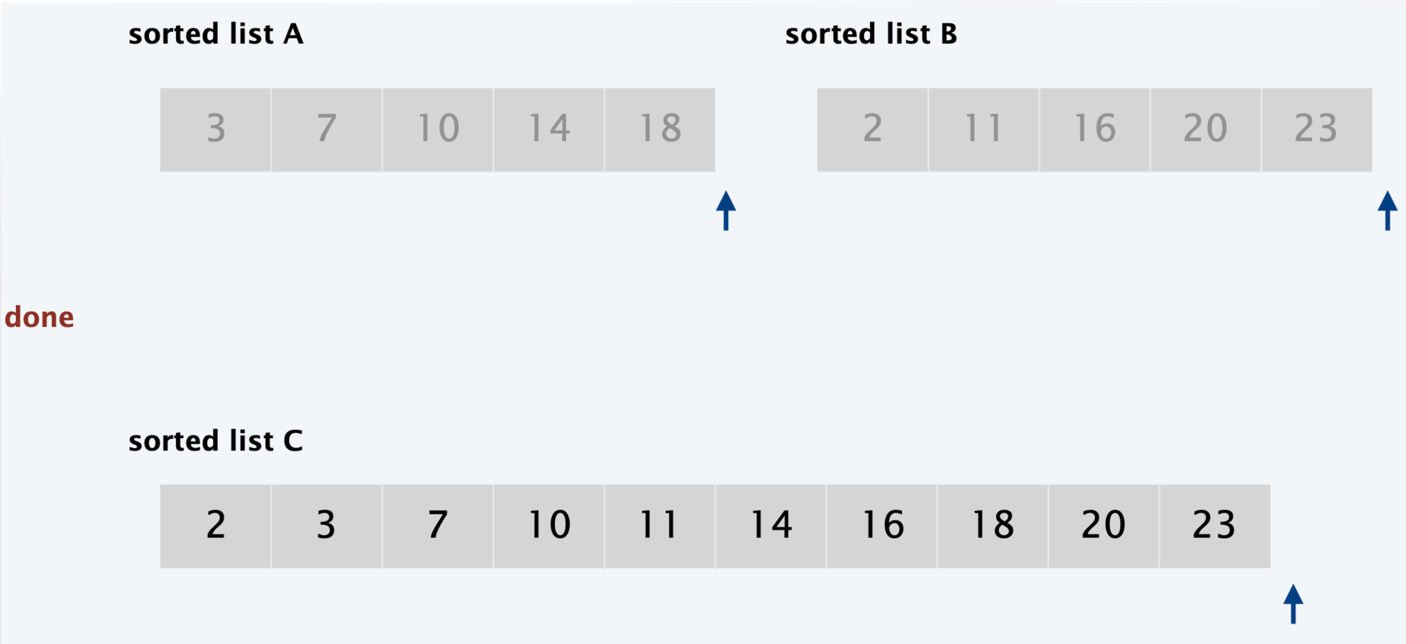
Algoritmos de Divisão e Conquista: Merge-Sort

- Dado duas listas ordenadas A e B, a combinação de ambas na lista C ocorre da seguinte maneira:



Algoritmos de Divisão e Conquista: Merge-Sort

- Dado duas listas ordenadas A e B, a combinação de ambas na lista C ocorre da seguinte maneira:



- Para analisar o tempo de execução do Merge-sort, vamos abstrair o seu comportamento no seguinte template, que descreve muitos dos algoritmos de divisão e conquista.

A.11:

1. ***Divida a entrada em duas peças de tamanho igual;***
2. ***Resolva os dois subproblemas separadamente por recursão;***
3. ***Execute o particionamento da entrada e combinação da solução final em $\boldsymbol{O(n)}$.***

- No Merge-sort, bem como em qualquer algoritmo deste tipo, precisamos de um caso base para a recursão, tipicamente um critério de parada quanto a entrada atinge um determinado limite.
- No caso do Merge-sort, vamos assumir o limite de $|L| = 1$, ou seja, quando a lista a ser ordenada possuir 1 elemento.
- Ao atingir esse limite, paramos a recursão e ordenamos os dois elementos por simples comparação.

- Considere um algoritmo que se enquadra no padrão de A.11, e permita que $T(n)$ defina o pior tempo de execução numa entrada de tamanho n.
- Supondo que n é par, o algoritmo gasta $O(n)$ para dividir a entrada em duas partes de tamanho $n/2$ cada.
- Depois o algoritmo gasta $T(n/2)$ para resolver cada parte.
- E finalmente depende $O(n)$ para combinar as soluções das duas chamadas recursivas.
- Então o tempo de execução $T(n)$ satisfaz a seguinte relação de recorrência:

A.12: Para alguma constante c

$$\boldsymbol{T(n) \leq 2T(n/2) + cn}$$

quando $\boldsymbol{n > 1}$, e

$$\boldsymbol{T(1) \leq c}$$

- A estrutura de A.12 é o formato típico de uma recorrência:
  - Existe uma inequação ou equação que limita $T(n)$ em termos de uma expressão envolvendo $T(k)$ para valores menores k.
  - Existe uma caso base que afirma que $T(n)$ é igual a uma constante quando n é constante.
  - Também podemos escrever a equação de A.12 como $\mathrm{T(n)} \leq 2\mathrm{T}(n/2) + O(n)$
  - Para mantermos a estrutura simples assumimos que n é par, caso fosse ímpar teríamos: $\mathrm{T(n)} \leq \mathrm{T}(\lfloor n/2 \rfloor) + \mathrm{T}(\lceil n/2 \rceil) + cn$. Porém, os impacto assimptóticos são irrelevantes.

- Também, podemos escrever A.12 da seguinte maneira:

A.12: Para alguma constante c

$$T(n) \leq 2T(n/2) + cn$$

quando $n > 1$, e

$$T(1) \leq c$$

OU

$$T(n) = \begin{cases} T(1) & se \ \ n = 1 \\ 2T(n/2) + cn & se \ n > 1 \end{cases}$$

A.12: Para alguma constante c

$$\boldsymbol{T(n)} = \begin{cases} \boldsymbol{T(1)} & \boldsymbol{se\ \ n = 1} \\ \boldsymbol{2T(n/2) + cn} & \boldsymbol{se\, n > 1} \end{cases}$$

- Observe que A.12 não define um limite assimptótico explícito na taxa de crescimento de T.
- A.12 apenas especifica $T(n)$ em termos de $T(n)$ quando os valores de n são cada vez menores.
- Para obtermos um limite explícito, precisamos resolver a relação de recorrência de forma que T aparece apenas do lado esquerdo da inequação.
- A tarefa de resolução de recorrências foi incorporada num grande número de sistemas de álgebra computacional e a solução para muitos problemas de recorrência pode ser encontrada de forma automatizada.
- Porém, é útil compreender o processo de resolução de recorrências e reconhecer quais recorrências levam a bons tempos de execução.
- A performance dos algoritmos de divisão e conquista está diretamente relacionada com as relações de recorrência.

Existem duas abordagens para a resolução de recorrências:

1. “Desenrolar” a recursão:
    - Consiste na contabilização dos tempo de execução para alguns níveis e identificar um padrão que pode ser repetido a medida que a recursão expande.
    - Depois somamos os tempos de execução de todos os níveis da recursão para chegarmos ao tempo total de execução.

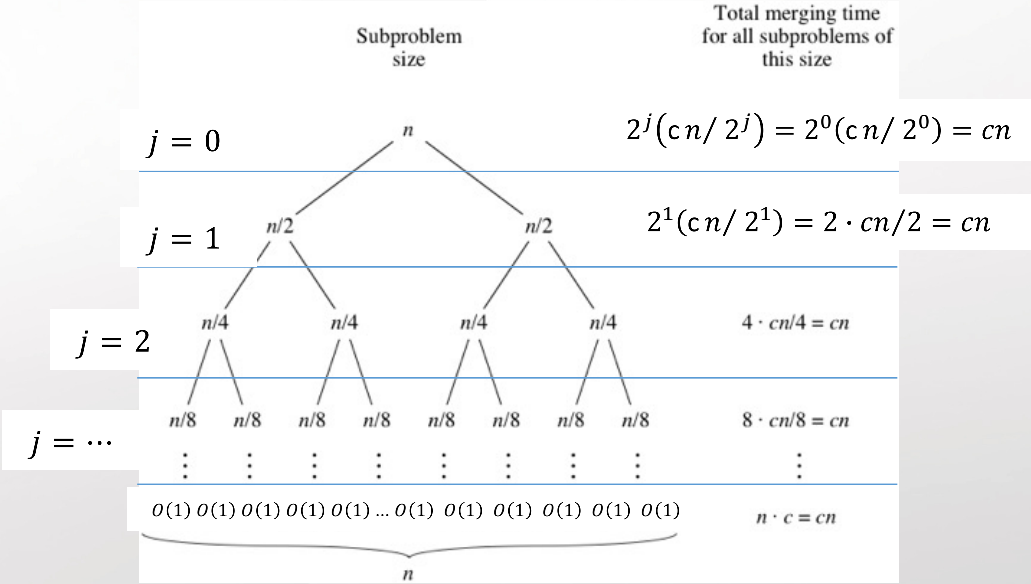
2. “Chute” inicial:
    - Começamos com um “chute” inicial da solução, substituímos na relação de recorrência e verificamos se funciona.
    - Formalmente, justificamos a inserção de solução inicial via um argumento de indução em n.

PUCRS ESCOLA POLITÉCNICA

Algoritmos de Divisão e Conquista:
"Desenrolar" a recursão

O "Desenrolar" da recursão possuí as seguintes etapas:

1. Começamos analisando os primeiros níveis do Merge-sort:
   - No primeiro nível da recursão, temos um único problema de tamanho n, que leva tempo cn mais o tempo gasto nas chamadas recursivas subsequentes.
   - No próximo nível, temos dois problemas de tamanho $n/2$. Cada um com um custo de $cn/2$ para um total de no máximo cn mais o tempo gasto nas chamadas recursivas subsequentes.
   - No terceiro nível, temos quatro problemas $n/4$, levando tempo $cn/4$ e custo total cn.
2. Identificando um padrão:
   - No nível j da recursão, o número de subproblemas tem o dobro de j tempos, de forma que temos 2^j.
   - Cada problema encolheu por um fator de 2^j, ou seja, cada problema tem tamanho $n/\ 2^j$ e um tempo total de $cn/\ 2^j$.
   - Level j contribui com um valor de no máximo $2^j\left(cn/\ 2^j\right) = cn$ no tempo de execução total.



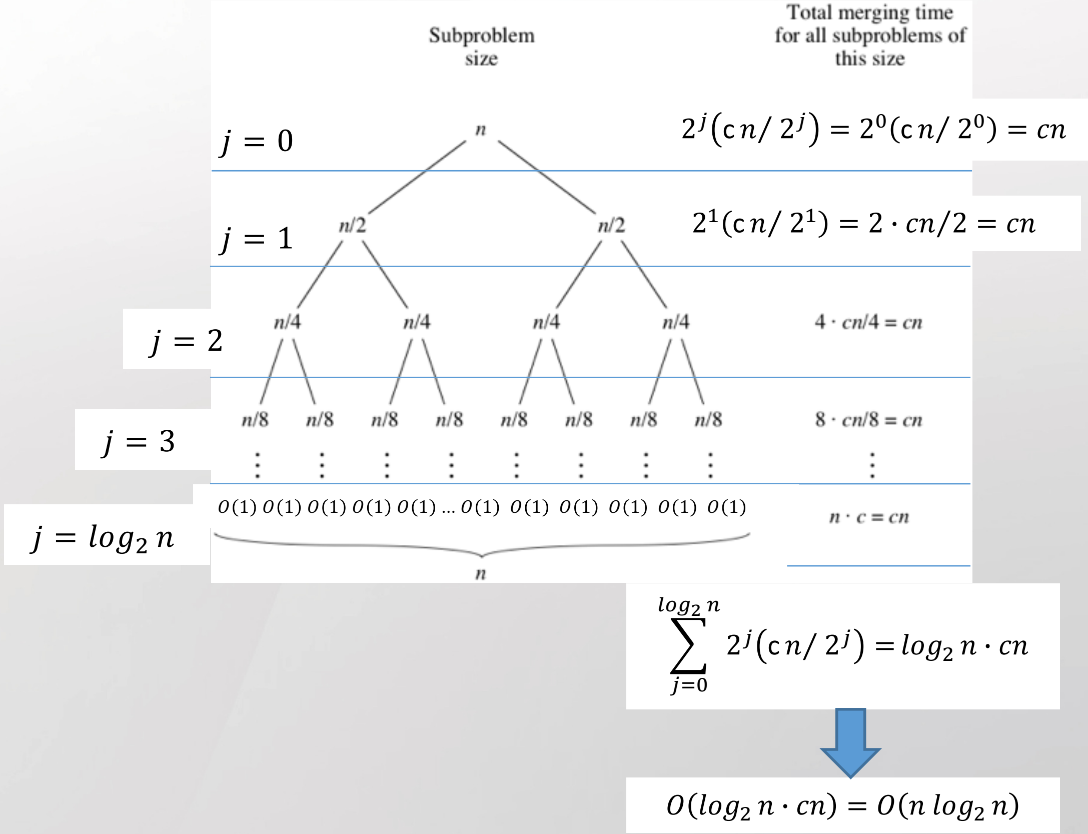
Algoritmos de Divisão e Conquista:

"Desenrolar" a recursão

O "Desenrolar" da recursão possuí as seguintes etapas:

3. Somando todos os níveis da recursão:
   - Descobrimos que a recorrência em A.12 tem a propriedade que o mesmo limite superior de cn é aplicável para o total de trabalho performado em cada nível, ou seja, cada nível é limitado por $O(n)$.
   - O número de vezes que a entrada n precisa ser particionada para atingir o valor de parada (fundo da recursão) 2 é $\log_2 n$.
   - Portanto, somando o trabalho cn sobre todos os $\log_2 n$ níveis da recursão, chegamos ao valor total de tempo de execução do Merg—Sort é $O(n \log n)$.

A.13: Qualquer função $T(\cdot)$ que satisfaça A.12 é limitada por $O(n \log n)$, quando $n > 1$.



Algoritmos de Divisão e Conquista:

"Chute" inicial

Na abordagem do "chute" inicial procedemos da seguinte forma:

- Suponha que temos um palpite de que $T(n) \leq cn\ log_2 n$ para todos $n \geq 1$ e queremos verificar se isto é verdadeiro.
- A afirmação funciona para $n = 1$, pois segundo o algoritmo e A.12, $\mathrm{T}(1)$ é resolvido em tempo constante, ou seja, $\mathrm{T}(1) \leq c$.
- Agora suponha por indução que $T(m) \leq cm\ \log_2 m$ é valida para todos os valores de m menores que n e queremos demostrar isso para $T(n)$.
- Fazemos isso escrevendo a recorrência para $T(n)$ e plugando na inequação $T(n/2) \leq c(n/2)\, log_2(n/2)$.
- Depois simplificamos a expressão notando que $log_2(n/2) = log_2(n) - 1$. O que nos leva ao seguinte resultado:

$$T(n) \leq 2T(n/2) + cn$$

$$\leq 2c(n/2)\, log_2(n/2) + cn$$

$$= cn[log_2(n) - 1] + cn$$

$$= (cn\, log_2(n)) - cn + cn$$

$$= cn\, log_2(n)$$

- O resultado define um limite para $T(n)$, assumindo que funciona para valores menores $m < n$, e isso completa o argumento de indução.

Algoritmos de Divisão e Conquista:

Outras Relações de Recorrência $(\boldsymbol{a > 2})$

- O algoritmo de Merge-sort utiliza 2 chamadas recursivas a cada nível de iteração $(a = 2)$.
- Este problema pode ser generalizado para mais do que duas chamadas recursivas por iteração $(a > 2)$.

A.14: Para alguma constante c

$$\boldsymbol{T(n) \leq aT(n/2) + cn}$$

quando $\boldsymbol{n > 1}$, e

$$\boldsymbol{T(1) \leq c}$$

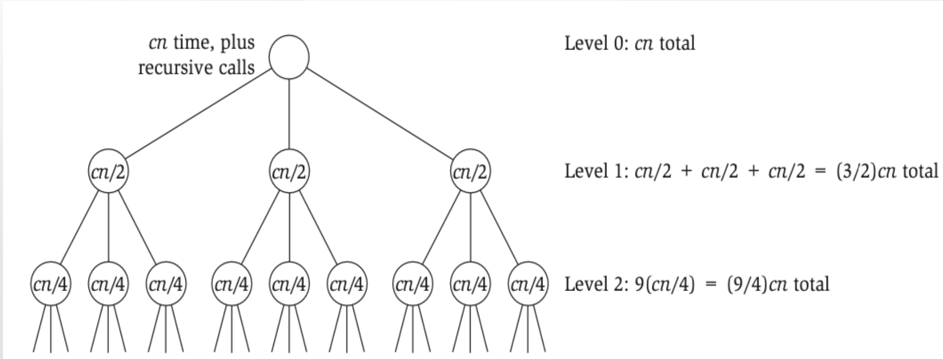
- Vamos resolver pelo "desenrolar da recursão" ou árvore de recursão.

Algoritmos de Divisão e Conquista:

Outras Relações de Recorrência $(a > 2)$

Vamos resolver o problema para um algoritmo com 3 chamadas recursivas por nível $(a = 3)$.

1. Começamos analisando os primeiros níveis:
   - No primeiro nível da recursão, temos um único problema de tamanho n, que leva tempo cn mais o tempo gasto nas chamadas recursivas subsequentes.
   - No próximo nível, temos q problemas de tamanho $n/2$. Cada um com um custo de $cn/2$ para um total de no máximo $(a/2)cn$ mais o tempo gasto nas chamadas recursivas subsequentes.
   - No terceiro nível, temos q^2 problemas $n/4$, levando tempo $(cn/4)cn$ e custo total$(a^2/4)cn$.
2. Identificando um padrão:
   - No nível j da recursão, temos a^j instâncias de tamanho $n/\ 2^j$.
   - Level j contribui com um valor de no máximo $a^j\left(\text{c}\,n/\ 2^j\right) = (a/2)^j cn$ no tempo de execução total.



Algoritmos de Divisão e Conquista:

Outras Relações de Recorrência $(a > 2)$

3. Somando todos os níveis da recursão:

- Com no caso de $a = 2$ temos $\log_2 n$ níveis de recursão, e um árvore com $\log_2 n + 1$ níveis no total.
- O trabalho total performado em todos os níveis é:

$$T(n) \leq \sum_{j=0}^{\log_2 n} \left(\frac{a}{2}\right)^j cn = cn \sum_{j=0}^{\log_2 n} \left(\frac{a}{2}\right)^j$$

- Isso é uma soma geométrica com razão $\boldsymbol{r = a/2}$, portanto a soma converge para:

$$T(n) \leq cn\left(\frac{r^{\log_2 n} - 1}{r - 1}\right) \leq cn\left(\frac{r^{\log_2 n}}{r - 1}\right)$$

- Como estamos interessados no limite assimptótico superior, é útil procurarmos apenas uma constante. Portanto, removemos $r - 1$ dos parênteses.

$$T(n) \leq \left(\frac{c}{r - 1}\right) n r^{\log_2 n}$$

- Agora precisamos definir o que $r^{\log_2 n}$ significa.
- Para isso utilizamos uma identidade, que afirma que para qualquer a > 1 e b > 1, temos que $a^{\log_2 b} = b^{\log_2 a}$.

$$r^{\log_2 n} = n^{\log_2 r} = n^{\log_2 (a/2)} = n^{\log_2 a - 1}$$

- Então:

$$T(n) \leq \left(\frac{c}{r - 1}\right) n \cdot n^{\log_2 a - 1} = \left(\frac{c}{r - 1}\right) n^{\log_2 a}$$

- O resultado acima é $\mathrm{O}\left(n^{\log_2 a}\right)$.

A.15: Qualquer função $T(\cdot)$ que satisfaça A.14 é limitada por $O\left(n^{log_2 a}\right)$, quando q > 2.